Synthesis Oriented Scheduling of Multiparty Rendezvous in Transaction Level Models

Vyas Venkataraman*, Di Wang*, Atabak Mahram*, Wei Qin*, Mrinal Bose†, Jayanta Bhadra†
* Electrical and Computer Engineering Dept.
Boston University; Boston, MA 02215, USA
† Freescale Semiconductor Inc.
Austin, Texas 78729, USA
{vyas,diwang,mahram,wqin}@bu.edu {mrianal.bose,jayanta.bhadra}@freescale.com

Abstract

Due to the large semantic gap between transaction level models and actual implementations, hardware synthesis based on system level models has been a great challenge. Aiming to close the semantic gap, we studied an approach that uses rendezvous to model communication. By allowing both conjunctive and disjunctive composition of rendezvous, the approach supports flexible communication patterns involving multiple processes. However, a practical issue of the model is the complexity of scheduling of multiparty rendezvous, which is NP hard in general. This paper proposes an efficient scheduling algorithm. It begins by encapsulating state transition information of processes into a relation graph. It then creates a tree that relates edge combinations. The tree is used to guide the scheduler at run time to search for schedulable sets. Experimental results prove that this algorithm improves our scheduler significantly. The algorithm lays the ground for the synthesis of the communication and synchronization circuitry for the system.

1. Introduction

As the complexity of modern systems grows, it necessitates an increase in the abstraction level of models used to describe them. Typically, a large system is maintained in at least two models, a transaction level model in a high level language such as C++, and a synthesizable implementation in a hardware description language at the register transfer level. This semantic gap leads to wasted productivity in duplication and a verification gap between the two models.

This paper is based on a modeling methodology utilizing rendezvous as the fundamental construct for interprocess communication. Rendezvous, commonly used in concurrent programming languages such as Ada [1], offer a safe and versatile mechanism for abstracting communication. Specifically, rendezvous enable two or more processes to communicate and synchronize. They support composition in both conjunctive and disjunctive manners. Their expressiveness allows many useful communication patterns to be modeled. This paper investigates the problem of scheduling and synthesizing rendezvous based hardware models.

The complexity of scheduling multiparty rendezvous is primary major barrier in using rendezvous. In theory, the complexity is known to be at least NP-complete [2]. However, in practical systems, the static relationships between all rendezvous help reduce scheduling complexity significantly. This paper presents an algorithm that explores these relationships. It begins by encapsulating state transition information into a relation graph. It then creates a tree that relates edge combinations. This tree guides the scheduler at run time in the search for schedulable sets. Simulation results show a significant improvement of efficiency over brute force approaches. Moreover, the resulting tree can be directly synthesized into hardware.

This paper is organized as follows. Section 2 quickly overviews rendezvous based modeling languages. In Section 3, we present our modeling approach and formulate the scheduling problem. Sections 4 and 5 describe the construction of the scheduling tree, and the actual scheduler based on the tree. Section 6 examines experimental results. Section 7 concludes the paper.

2. Related Work

The main focus of high level modeling is on abstracting communication between basic modules. Traditional software approaches are comfortable in using function calls for communication, giving rise to transaction level modeling via System-C [3]. These port based function calls do not require special scheduling support and have loose semantic requirements. However, there are no primitives to model complex communication patterns, such as barrier synchronization of multiple processes. Most importantly, the lack of a theoretical foundation prevents the easy synthesis of such models.

A popular communication construct in concurrent programs is rendezvous, a synchronous handshaking mechanism among processes. In its simplest bi-party form, two processes await each other at a point where they expect to communicate. When both are present, the rendezvous occurs, allowing them to continue.

Various flavors of rendezvous have been studied in process calculi such as CSP [4] and CCS [5] and in concurrent programming languages such as Ada [1] and Occam [6]. More recently, rendezvous style communication has been used in languages such as Ada-C [7], Haste [8] and SHIM [9], which explicitly target the modeling of hardware-related systems. Using a similar method to [2], we can classify the concurrent languages...
according to the following properties of their respective rendezvous - multipartness, variability of participants, and rules of composition. Multipartness refers to the number of parties a single communication involves. Variability refers to whether the participants in a communication must be a fixed set of processes or not. Composition rules decide on ways of combining different communications. Disjunctive composition allows one out of a set of rendezvous to be chosen to make progress. Conjunctive composition is the case where a few rendezvous must occur simultaneously in order to make progress. Table 1 classifies the languages according to these aspects.

### Table 1. Features of Concurrent Languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Multiparty</th>
<th>Variability</th>
<th>Disjunction</th>
<th>Conjunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Occam</td>
<td></td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Handel-C</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Haste</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>SHIM</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In general, scheduling in the presence of only multipartness and disjunction is relatively easy. Individual processes can progress independently using only local knowledge. On the other hand, variability and conjunctive composition generally require the use of arbitration and coordination. Variability enables competition, implying the need for arbitration. Conjunctive composition teams up multiple processes, implying the need for coordination. However, the surveyed languages avoid multiparty scheduling by placing restrictions on their use of rendezvous. For example, Handel-C requires that no two processes should write to the same channel (rendezvous) simultaneously, leaving the burden of arbitration to designers. Haste supports limited arbitration of channels, but does not coordinate the processes when variability and composition coexist, leaving many deadlock hazards for designers to handle.

### 3. Problem Statement

The modeling methodology under study resembles Haste in the supported properties of rendezvous, but is intended for use at higher abstraction levels. It supports two types of rendezvous, a variable participant, bi-party rendezvous that carries bidirectional data flow, and a fixed participant, multi-party barrier rendezvous. It supports both conjunctive and disjunctive composition of rendezvous. A system model consists of a network of processes connected by rendezvous. The processes are expressed as state diagrams, where state transition edges are annotated with semantic actions, the rendezvous required for the transitions to occur, as well as Boolean expressions as guards. Processes synchronize on transition edges labeled with common rendezvous.

The example network shown in Figure 1 consists of 2 stages of an elastic pipeline [10] with one producer and a consumer. An elastic pipeline is a FIFO like structure that is used to couple a producer and a consumer that have a mismatch in speeds, having a variable latency, depending on the number of data elements already in the pipeline. In Figure 1, the two processes stage₀ and stage₁ are the two stages of the pipeline which connects the producer process to consumer process. The system contains 3 bi-party rendezvous $Prop₀$, $Prop₁$, and $Prop₂$, each of which have 2 roles labeled “+” and “-”. In order for a rendezvous to occur, exactly one of each role label must participate simultaneously. For example, for $Prop₀$ to occur, a $Prop₀+$ and a $Prop₀-$ label must participate. Thus, the simultaneous transitions of the producer along edge 1 and stage₀ along edge 3 will satisfy the requirement for $Prop₀$ to occur. In addition, several rendezvous may occur jointly due to conjunctive composition. For example, joint transitions of edges 1, 2 and 7 imply the joint occurrence of rendezvous $Prop₀$ and $Prop₁$. For brevity, Figure 1 only shows rendezvous labels on transition edges but omits the dataflow and the semantic actions. Though they have identical labels, edges 2 and 5 handle data differently. On edge 2, the input received is directly forwarded; while on 5, the input is stored into a local register.

To quantify the importance of the rendezvous, we associate each rendezvous with an integer weight. The scheduling problem can be stated as follows: Given a process network and its global state, find the set of rendezvous that can occur with the maximum total weight. The problem can be formulated as a pseudo-Boolean problem (a.k.a. 0-1 integer programming), which is known to be NP-hard.

We first define a Boolean variable $x_i$ for each rendezvous, whose value indicates its occurrence. We also define $f_i$ for each rendezvous label $l$ (the appearance of a role of a rendezvous on an edge), whose value indicates the choice of $l$ should the corresponding rendezvous occur. For each edge $e$, we define a Boolean variable $x_e$, whose value indicates the choice of the edge

![Figure 1 Example process network](image)
by its own process for state transitioning. We can then define the following constraints.

1. **Rule for bi-party rendezvous.** The occurrence of a bi-party rendezvous involves exactly a “+” label and a “–” label. Therefore, for every bi-party rendezvous and its labels \( \{r_i^+, r_i^-\} \), \( 0 \leq r_i = \sum r_i^+ = \sum r_i^- \leq 1 \);

2. **Rule for multiparty barrier.** The occurrence of a multiparty barrier rendezvous involves all its roles. Therefore, for every multiparty barrier and its labels \( \{r_j^i\} \) for role \( j \), \( x_i = \sum x_i^j \).

3. **Rule for conjunctive composition.** If an edge is taken for transition, then all rendezvous labels on its edge should be chosen to occur. Therefore, for every edge and all its labels \( \{r_i, x_i\} \).

4. **Rule for disjunctive composition.** If a process is able to make progress, it should do so along only one edge from the current state. Therefore, for every active state and its outgoing edges \( \{e\} \), \( \sum x_i \leq 1 \).

The goal of the problem is thus to maximize \( \sum w_i x_i \), where \( w_i \) is the integer weight of \( r_i \).

The above problem needs to be solved by a central scheduler for each step during execution. The NPhardness renders models with a large number of rendezvous impractical. This is further complicated by the existence of data flow and guard conditions on the state transition edges, especially when some conditions depend on the data values carried by the rendezvous.

In practice, however, not all rendezvous are related. The intuition behind this is that in a hardware-oriented model, if several rendezvous are closely related to each other and require the scheduler to arbitrate, they are probably connected by some combinational logic. Since the lengths of combinational paths are limited in most practical systems, subsets of related rendezvous should not be large. Moreover, even when two rendezvous are related to each other, their relationship can be analyzed to pick out an occurrence pattern.

### 4. Algorithm Description

The use of both variability and conjunctive parallelism causes the proposed rendezvous based modeling methodology to have significant scheduling complexity. The algorithm starts by creating a relationship graph between all transition edges in the system. This graph, called transition relation graph (TRG), is then used to produce a scheduling tree (ST), which guides run time scheduling. The leaf nodes of the tree contain minimal sets of transition edges that must occur jointly.

#### 4.1. Transition relation graph (TRG)

The purpose of the transition relation graph is to establish occurrence relationships between different state transition edges. The construction algorithm for the TRG accepts a system description, consisting of a network of sequential processes, each represented as a state transition diagram. It builds a graph where the vertices are state transition edges. Each vertex is annotated with a set of rendezvous role labels belonging to the corresponding state transition edge. The TRG’s edges relate all state transition edges in all processes in the system. In order to express all these relationships, we define three kinds of TRG edges, a related edge (represented graphically as a solid line), a dynamic mutual exclusion (DME) edge (represented by a black dashed line) and a static mutual exclusion (SME) edge (represented by a red dashed line). The main difference between DME and SME edges is that the SME relationship can be easily resolved using run time state information, while DME requires possible arbitration.

If two transition edges in the same process have the same set of labels, but different source states, we merge them into one vertex. As a process can only be in a single state at any instant, the ambiguity introduced by such merging can always be resolved using the state information at run time. After that, we examine all vertices pairwise.

1. If the vertices belong to the same process and have the same source state, draw a DME edge between them. This edge captures the disjunctive relationship of the transition edges from a single state – the choice of one particular edge precludes the other.
2. If the vertices belong to the same process but have different source states, draw an SME. This edge captures the mutual exclusion of the states of a process, since a process can be at one state at a time.
3. If the vertices belong to different processes, but share common rendezvous role labels, draw a DME edge between them. This edge captures the competition of the processes for the same role of a rendezvous. As only one participant can fulfill a role, the choice of one vertex for a rendezvous role precludes the other.
4. If the vertices belong to different processes, but have different roles of the same rendezvous, draw a related edge between them. A related edge captures the possibility of joint occurrence of two vertices.

We note that we may have at most as many vertices in the TRG as transition edges in the original system. The number of edges in the TRG depends on the construction of the network, the number of labels of a role of a rendezvous, and the number of transition edges in a state in the system.

In the given example, we can begin by converting all edge labels into vertices, obtaining the set of vertices. Now we observe that in the stage, process, the two vertices 2 and 5 have identical rendezvous role labels and are in the same process but have different source states. We can then merge the two into a single vertex. Similarly, we can merge vertices 6 and 9 of stage. Now we can draw DME (black) and SME (red) types of edges be-
tween all vertices in a process, as well as related edges across processes. This gives us Figure 2.

![Figure 2 Final TRG](image)

**4.2. Scheduling Tree (ST)**
We first define a few helper functions below.
- `labels(v)` returns all rendezvous labels associated with vertex v.
- `isComplete(V)` accepts a set of vertices and verifies that this set is complete so that all its transition edges can jointly occur. In other words, the set contains exactly one label for each role of each included rendezvous.

We then define a recursive algorithm to create a tree based on the TRG. The tree contains three types of nodes. It exploits parallelism between disconnected components in the TRG through the use of _join nodes_. In the absence of natural parallelism, we can partition the TRG through the use of _decision nodes_, into two cases – one where a particular transition edge occurs, called the taken branch and the other where it does not, called the untaken branch. The _leaf nodes_ in the ST represent minimal sets of state transition edges that may occur jointly. A special type of leaf _Φ_ is encountered when the set of transition edges fails the completeness test done by the _isComplete_ function below. This is typically due to a conflict of the decisions made by the ancestor nodes. The _Φ_ leaves will be ignored when the scheduling tree is used.

The tree building algorithm receives a TRG _G_ and _V_\(\_\text{pin}\), a set of preselected vertices. Initially, _G_ is the full TRG, while _V_\(\_\text{pin}\) is empty. While recursion deepens, _G_ shrinks, _V_\(\_\text{pin}\) grows, and the three types of nodes are inserted into the tree.

The algorithm involves one heuristic when selecting decision nodes. It selects the vertex with the largest mutual exclusion degree as the decision node. This heuristic is based on the intuition that branching on a vertex that will result in the immediate deletion of the largest number of mutually exclusive neighbors in the taken branch, and as such, will most quickly shrink the size of the TRG, and divide it into disconnected components. Note while considering the connectivity of _G_, we can safely ignore SMEs since the mutual exclusion relationship of the vertices connected by an SME is preserved by runtime state information. On the other hand, DMEs imply decision making and thus must always be considered.

The heuristic only affects the shape of the tree. Regardless the heuristic used, the algorithm will always produce exactly the same set of unique leaf nodes. The number of nonempty leaf nodes is equivalent to the number of minimal sets of transition edges that may jointly occur. The set of leaf nodes is determined by the ways any particular rendezvous can occur and on how rendezvous are joined together via conjunctive composition. In the worst case, the number is exponential. In practice, the number of leaf nodes expresses the complexity of the communication in the system, which is well bounded for practical designs. For the example in Figure 1, we obtain the tree shown in Figure 3.

The tree building algorithm is fully deterministic. For a particular subgraph of the TRG, it will always yield the same subtree. Thus, as an optimization method, we use a hash table to keep track of sub-trees created. Whenever _BUILD_ is invoked on a previously seen _G_, it will return then previous result.

```plaintext
BUILD(G, V_\text{pin})
if hash[G] exists, return hash[G]
else
    G_\text{c} = G \setminus \text{all SME edges}
    if G_\text{c} is disconnected so that G_\text{c} = \cup G_i,
        j = new join node
        foreach G_i,
            add BUILD(G_i, V_\text{pin}) as child to j
        hash[G] = j
        return j
else if no SME or DME edge exists
    if (isComplete(V(G)))
        l = new leaf node (V(G))
        else
            l = new leaf node Φ
        hash[G] = l
        return l
else
    find v ∈ V(G) of maximum ME degree
    d = new decision node (v)
    G\’ = fixUp (G, ME neighbors of v, V_\text{pin} \cup \{v\})
    G\’’ = fixUp (G, \{v\}, V_\text{pin})
    add BUILD(G\’, V_\text{pin} \cup \{v\}) as taken child to d
    add BUILD(G\’’’, V_\text{pin}) as untaken child to d
    hash[G] = d
    return d

fixUp (G, V_\text{det} , V_\text{pin}) // repeatedly delete invalid vertices
while V_\text{det} ≠ ∅
    v = pop V_\text{det}
    if (v ∈ V_\text{pin}) // conflict, selected cannot be deleted
        return Φ
    else
        G = G \setminus v
        foreach l ∈ labels(v)
            if (G contains no more l)
                V_\text{det} ⊇= \{v \mid \text{labels(v) contains other roles of l}\}
        return G
```

244
5. Scheduling
The purpose of a run-time scheduling algorithm is to select the best occurrence set of state transition edges to make system progress. We implemented two types of schedulers, brute force schedulers enumerating all combinations of rendezvous, and a ST-based scheduler that traverses the ST in a single pass.

![Figure 3 Example of ST](image)

5.1. Brute Force Scheduling (BFS)
A simplistic scheduler can exhaustively check all combinations at run time for each simulator step. The BFS algorithm accepts a set of rendezvous, and the current global state. It first filters the rendezvous based on the state information, removing those that cannot occur because their associated transition edges are not incident to the current states, or because the numerical guard conditions are false. It then exhaustively tests every subset of the remaining rendezvous labels. In general, for each bi-party rendezvous, the algorithm checks \( m \times n \) occurrence possibilities, if it has \( m \) labels of role “+” and \( n \) labels of role “−”, and \( 1 \) non-occurrence possibility. Similarly, a barrier has \( \Pi b \) occurrence possibilities where \( b \) is the count of its labels of role \( j \), and \( 1 \) non-occurrence possibility. Thus, the worst-case time complexity of BFS is \( \Pi(m+1)\Pi(\Pi b+1) \), where the first term is for bi-party rendezvous, and the second for barriers. If we consider the example system from Figure 1, we can see that there exist a total of 15 transition edge occurrence possibilities using BFS.

A previous optimization to the BFS algorithm [11] attempted to use the static relationship of rendezvous reduce the search space of BFS. This algorithm, called Guided BFS (GBFS), categorized the relationship among rendezvous into mutual exclusion, implication, equivalence, independence, and dependence. It significantly improves the efficiency of BFS in most cases. However, when the relationship cannot be clearly categorized, such as in the elastic pipeline example in Figure 1, the approach degrades into BFS. Moreover, it is not convenient to synthesize GBFS into hardware.

5.2. Scheduling with ST
Using the ST we can simplify the problem of scheduling as a single pass traversal of the tree. For software simulation, the scheduler is implemented as a depth first search function that accepts \( T \), the scheduling tree, and \( S \), the current global state of the system. It returns the best set of transition edges that may jointly occur. At run time, the transition edges that are not incident to the current states of the processes can be ignored. This pruning helps resolve all merged nodes, eliminate the taken children of the decision nodes corresponding to those pruned transition edges, and eliminate the leaf nodes containing them. Moreover, as every leaf node contains a complete set of transitions, all data flow expressions and numerical conditions can be evaluated in parallel for each leaf node independently. Any leaf node containing a false numerical condition on any of its transition edges can be removed from consideration. After pruning, we traverse the tree to find the best set of leaf nodes to activate, based on the total weight objective. At a decision node, we recursively evaluate both subtrees and select the heavier set of leaf nodes returned by them. At a join node, we traverse all children and combine their returned sets of leaf nodes. At a leaf node, we return the leaf node itself if it is not pruned, or return an empty set if it is. In the example of Figure 3, we only need to examine 6 unique leaf nodes in parallel, and select or combine them when moving up toward the root. It is meaningful to examine the time complexity of the improved algorithm, which is the same as a depth first traversal of the tree. For a software implementation, this is proportional to the number of nodes in ST. In a hardware implementation, various paths in the scheduling tree can be evaluated in parallel. The time complexity there is thus proportional to the height of the tree.

5.3. Synthesis considerations
When creating an efficient scheduler in combinational hardware, it is of great importance that the scheduler be free of loops. Simple loops without loop-carried dependence may be eliminated by unrolling. So in theory, the BFS algorithm can be unrolled into a massively parallel array of combinational blocks, one for every occurrence pattern of all rendezvous. A large priority multiplexer then selects the ready pattern with the largest weight. However, such a scheduler would be too expensive and slow in practice. The GBFS algorithm greatly reduces the number of patterns to check. However, it is still expensive when the relationship among rendezvous is not easily identifiable.

On the other hand, the ST based scheduling algorithm is amenable to hardware implementation. The pruning
function supplies a condition guard signal for each vertex based on the current state. Every unique leaf is implemented as a combinational circuit block, which produces a ready signal as the conjunction of all pruning guards of the vertices in the leaf. The weights of each leaf are statically known and can be encoded as signals to priority multiplexers to reproduce the weight-based selection policy in the decision nodes of the tree. At the root level, a set of ready leaves are enabled, and their semantic actions then take place. As all checks are performed in a single pass traversal of the scheduling tree, the whole tree can be directly mapped to a combinational circuit if timing constraints can be met. The synthesis of the semantic actions of leaf nodes can utilize the traditional behavioral synthesis algorithms.

In summary, the importance of the work is the introduction of a synthesis-friendly scheduler of rendezvous-style inter-process communication in the presence of multipartiness, variability, disjunction and conjunction. To our best knowledge, no previous publication has a solution to the same problem.

6. Results
We implemented our modeling methodology in a custom framework that includes a description language to describe rendezvous based processes. We used some practical models to evaluate the proposed algorithm. The models listed in Table 2 are at different abstraction levels, demonstrating the flexibility of the methodology.

<table>
<thead>
<tr>
<th>Processes</th>
<th>Rendezvous</th>
<th>Transition Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEX</td>
<td>28</td>
<td>26</td>
</tr>
<tr>
<td>MIPS32</td>
<td>21</td>
<td>16</td>
</tr>
<tr>
<td>JPEG</td>
<td>31</td>
<td>41</td>
</tr>
<tr>
<td>EP 4</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>EP 8</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>EP 16</td>
<td>18</td>
<td>17</td>
</tr>
</tbody>
</table>

The first model PEX is a transaction level model of a PCI Express [12] topology, consisting of a root complex, a switch and two endpoints. The rendezvous model packet transactions in the data link layer. The second model tested, MIPS32 is a micro-architectural model of a 32 bit processor supporting a subset of the MIPS instruction set. Its rendezvous model data transfer between registers. The third model is a data flow model of the JPEG compression algorithm using rendezvous. The rendezvous model data flow communication. Finally, an EP n is an elastic pipeline with n stages, along with a producer and consumer.

We compare the benefit of using the ST to guide the scheduling algorithm to BFS and GBFS, by examining the average number of edge occurrence combinations to test per step of simulation, as shown in Table 3.

7. Conclusions
This paper presents a novel synthesis friendly scheduler of rendezvous style inter-process communication involving multipartiness, variability and both types of composition. Simulation results demonstrate large speedups in practical designs. The approach is applicable to existing languages such as Haste. Future work includes the refinement of the heuristic used so as to reduce the height of the scheduling tree and its overall size, as well as the implementation of a synthesis engine that converts system descriptions modeled using this rendezvous based methodology to hardware circuits.

<table>
<thead>
<tr>
<th>Processes</th>
<th>BFS</th>
<th>GBFS</th>
<th>STS</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEX</td>
<td>416286</td>
<td>653</td>
<td>17</td>
</tr>
<tr>
<td>MIPS32</td>
<td>371250000</td>
<td>30</td>
<td>22</td>
</tr>
<tr>
<td>JPEG</td>
<td>6563</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>EP 4</td>
<td>28</td>
<td>28</td>
<td>3</td>
</tr>
<tr>
<td>EP 8</td>
<td>2188</td>
<td>2188</td>
<td>5</td>
</tr>
<tr>
<td>EP 16</td>
<td>143489080</td>
<td>143489080</td>
<td>9</td>
</tr>
</tbody>
</table>

8. Acknowledgements
We thank Freescale Semiconductor for generously supporting this work via SRC grant 1622.

9. References